An Executable Calculus for Service Choreography

Paolo Besana[1] and Adam Barker[2]

[1]University of Edinburgh, [2]University of Melbourne

Abstract The Lightweight Coordination Calculus (LCC) is a compact choreography language based on process calculus. LCC is a directly executable specification and can therefore be dynamically distributed to a group of peers for enactment at run-time; this offers flexibility and allows peers to coordinate in open systems without prior knowledge of an interaction. This paper contributes to the body of choreography research by proposing two extensions to LCC covering parallel composition and choreography abstraction. These language extensions are evaluated against a subset of the Service Interaction Patterns, a benchmark in the process modelling community.

1 Introduction

A core challenge in today's ever connected world is in combining distributed resources on-demand to perform coordinated tasks. Coordination of distributed resources can be achieved through the use of workflow technologies. Workflow specifications defined using standard orchestration languages such as the Business Process Execution Language[1] typically facilitate statically defined workflows to be enacted by a centralised workflow engine. The resulting workflow specifications can be brittle due to the highly dynamic nature of distributed resources and scalable only to a point [1]. Dynamic composition of resources around tasks can address these issues.

The OpenKnowledge project[2] has produced a framework providing the middleware that assorted peers can use to interact within an open system. Peers interact following protocols, named *interaction models*, that define their externally observable behaviours. These protocols provide a global view of the interactions, and therefore fit into the definition of choreography, as opposed to the one of orchestration, considered to be a description of coordination from the perspective of a single process. The protocols are written in the Lightweight Coordination Calculus (LCC), a compact choreography language based on process calculus.

This paper introduces the LCC syntax, corresponding OpenKnowledge framework (Section 2) and discusses its current limitations and possible workarounds (Section 3). We propose two extensions to the LCC language (Section 4) and evaluate these changes by analysing the representations of two interaction patterns (Section 5). Related work is discussed (Section 6) in the context of our language extensions. Finally our contributions are summarised (Section 7).

[1] www.oasis-open.org/committees/wsbpel/

[2] www.openk.org/

2 OpenKnowledge and LCC

The OpenKnowledge kernel [7] has been designed with the goals of light weightness and compactness. The core concept is the set of shared *interaction models*, enacted by participants, called *peers*, that play *roles* within them. Interaction models are written in the *Lightweight Coordination Calculus* (LCC) [5], a compact, executable choreography language based on process calculus.

An interaction model in LCC is a set of role clauses. Clauses can refer to *entry-roles*, which participants initially assume, and *auxiliary-roles*, that can be reached only from other roles. Participants in an interaction take their *entry-role* and follow the unfolding of the clause specified using a combination of the sequence operator ('*then*'), or choice operator ('*or*') to connect messages and changes of role. A participant can take several roles during an interaction and can recursively take the same role (for example when processing a list). Messages are either outgoing to ('⇒') or incoming from ('⇐') another participant in a given role. Message input/output or change of role is controlled by constraint satisfaction.

```
a(auctioneer(Product, Bidders), A) ::
 a(caller(Product, Bidders), A)
 then a(waiter(Bidders, curwinner(nul, 0), Winner)
 then sold(Product, Price) ⇒ a(bidder, WB) ← curwinner(WB, Price) = Winner

a(caller(Product, Bidders), A) ::
 null ← Bidders = [] % no bidders left
 or ( invite_bid(Product) ⇒ a(bidder, BH) ← Bidders = [BH|BT]
    then a(caller(Product, BT), A) % recursion )

a(waiter(Bidders, Bids, curwinner(WinBidder, WinBid), Winner) ::
 null ← allarrived(Bids, Bidders) and Winner = curwinner(WinBidder, WinBid)
 or null ← timeout() and Winner = curwinner(WinBidder, WinBid)
 or ( bid(Offer) ⇐ a(bidder, B) then
      ( a(waiter([B|Bidders], curwinner(B, Offer), Winner) ← Offer > WinBid
        or a(waiter([B|Bidders], curwinner(WinBidder, WinBid), Winner) ) )
 or a(waiter(Bidders, curwinner(WinBidder, WinBid), Winner) ← sleep(1000)

a(bidder, B) ::
 invite_bid(Product) ⇐ a(caller, A)
 then bid(Product, Offer) ⇒ a(caller, A) ← bid_at(Product, Offer)
 then sold(Product, Price) ⇐ a(auctioneer, A)
```

Figure 1. Auction protocol, % represents a comment.

Figure 1 shows an interaction model for an auction. The interaction starts with the `auctioneer` role that receives the product to sell and the list of bidders as an input parameters. It immediately changes its role to `caller` passing in the list of bidders and the product to sell. The `caller` recurses over the list of

bidders in `Bidders`. If the list of peers is empty, it returns to the calling role; otherwise, it sends the `invite_bid` message to the peer at the head of the list and recurses over the remaining peers. Once all the messages are sent, the peer takes the `waiter` role, passing the list of bidders. The parameter `Winner` is an output parameter, and its value is set when the role `waiter` ends. The `waiter` role first checks if all the replies have arrived or if the period has timed out: if one of these two conditions is true, then it assigns the current winner as the final winner. Otherwise, it checks if there is a message in the incoming queue. If there is an offer and it is higher than the current highest offer, it recurses making the current bidder the current winner, otherwise it simply recurses. If there is no offer in the queue, then it waits for a second and recurses.

Symmetrically, the `bidder` receives the request to bid, and sends the offer. It may then receive the `sold` message if the offer was successful. If unsuccessful, the framework will signal the end of interaction. Through this pattern (an implementation of the *synchronisation* pattern [6]) asynchronous message reception is possible: messages are received in any order, and bidders act independently.

3 Limitations

While developing interactions for the various scenarios, we encountered limitations in the current version of OpenKnowledge. It was often possible to find workarounds but these ad-hoc solutions lacked generality and clarity. The limitations can be divided in two categories: design and execution. We will describe two design-time limitations: the impossibility of representing different levels of abstraction in a clean way and the lack of a parallel operator.

In LCC, a single interaction model includes all activities and messages at all levels of abstraction. The only abstraction available is provided by roles, that have to belong to the same interaction model. One possible work-around, applied in various cases in the testbeds used for evaluating OpenKnowledge [8], is for a peer to start a new interaction from within a constraint. However, this solution has two drawbacks: starting a new interaction is the action of single peer, of which other peers are not aware. This makes it hard to include participants involved in the first interaction into the sub-interaction. It is also a brittle solution, as it is not possible to specify how constraints are solved by peers.

Another important limitation is the lack of a parallel operator. Parallel operations, as described in the *parallel split* pattern [6], can be obtained by sending a sequence of messages to a set of roles waiting for them. Sending a message is a non-blocking operation (we saw before that it requires only to insert the message in a queue) so, from the perspective of the `auctioneer`, the operations in the bidders are started nearly simultaneously. In the example the replies from the bidders are merged back by the `waiter` role.

However, in this specific case we know the number of parallel operations (that is, the number of bidders) before the start of the interaction. In the general case it is not possible to know in advance how many parallel operations need to be performed: if peers have to be bound to these roles in advance it is not possible to increase their number once the interaction has started. Moreover, it may not be clear who should perform these roles.

4 Proposed Design Extensions

4.1 Scene Operator

To address the lack of an abstraction mechanism and to maintain at the same time clarity at design-time, we introduce the concept of *scenes*, which are abstractions of interaction models. In turn interaction models implement scenes. We also introduce the new operator `scene(scenename, role)`, that defines the execution of a role in another scene.

`a(auctioneer(Product), A)` ::
`a(caller(Product, Bidders, Winner), A)` ← `getPeers("bidder", Bidders)`
then `sold(Product, Price)` ⇒ `a(bidder, WB)` ← `curwinner(WB, Price) = Winner`
then $scene(payment, a(payee(Price), A))$

`a(bidder, B)` ::
`invite_bid(Product)` ⇐ `a(caller, A)`
then `bid(Product, Offer)` ⇒ `a(caller, A)` ← `bid_at(Product, Offer)`
then `sold(Product, Price)` ⇐ `a(auctioneer, A)`
then $scene(payment, a(payer(Price), B))$

Figure 2. Extending the auction with scenes.

An interaction models makes no assumption of how the scene will be performed: the operation has to be matched to another interaction model implementing the scene. This has to be performed by the enactment framework. A scene can succeed or fail, like a standard interaction model. Figure 2 shows how the auction protocol could be modified to include scenes. Once the winning bidder has been alerted, both the auctioneer and the bidder go into the *payment* scene, respectively in the `payee` and `payer` roles. The requirement is that all the peers in the calling interaction model that have encountered the same scene invocation are registered to participate in the run of the matched interaction model. The new scene can also include other roles and other participants. The addition of scenes does not influence the correspondence between LCC and π-calculus: at run-time the operation is equivalent to a normal role change. What changes is how the role is located and matched.

The use of scenes allows the creation of hierarchies of scenes at different levels of abstraction, in which the root is itself a scene. At each level, scenes are implemented by interaction models, that can contain other scenes, implemented by further interaction models and so forth.

4.2 Parallel Operator

The parallel operator we introduce here focuses around the operation of role change. While in the current version of LCC the role change operator inside a clause is always a sequential operation, we distinguish between two different role calls, one for blocking and one for non-blocking execution of a role clause: `b:a(type,ID)` and `nb:a(type,ID)`.

The current role call is blocking: the execution of the calling role is halted, the called role is executed, and when it terminates the caller resumes. The non-blocking call corresponds to spawning a new role in a parallel process: a role can spawn a new role, executed by the same peer. The spawned process has its own process identifier and its own incoming and outgoing message queues, like a normal participant.

```
a(auctioneer(Product, Time), A) ::
 b : a(caller(Product, Bidders), A) ← getPeers("bidder", Bidders)
 then nb : a(timer(Time), G) then
 then b : a(waiter(Bidders, curwinner(nul, 0), Winner)
 then sold(Product, Price) ⇒ a(bidder, WB) ← curwinner(WB, Price) = Winner

a(timer(Time), T) ::
 timeout ⇒ a(waiter, A) ← wait(Time)

a(waiter(Bidders, Bids, curwinner(WinBidder, WinBid), Winner) ::
 null ← allarrived(Bids, Bidders) and Winner = curwinner(WinBidder, WinBid)
 or ( bid(Offer) ⇐ a(bidder, B) then ... )
 or timeout ⇐ a(timer, T)
```

Figure 3. Auction interaction model with `timer` role.

While in a blocking call, it is possible to have both input and output parameters; in non-blocking calls, to avoid concurrency issues in accessing the parameters, the called role can only have input parameters. In the blocking role call the variable containing the process identifier contains the caller process, while in the non-blocking role call it is instantiated with the process identifier of the newly created process. To avoid zombie processes, the spawned role maintains a link with the spawner: if the spawner terminates, the spawned also terminates.

An interesting use of the parallel role is for timers. Using a parallel operator the auctioneer can start a parallel timer role that waits a finite amount of time and then sends a message. For instance, in Figure 3 the auction protocol is modified to include a timer, that after a fixed amount of time sends a message to the `waiter`. The `waiter` receives the message if some bidder has not replied before the deadline.

5 Evaluation

In order to evaluate our proposed extensions, we discuss how a subset of the interaction patterns described in [3] can be represented in the extended version of LCC. While all patterns that use time-frames can benefit from the introduction of the non-blocking role change, we will analyse in detail two of these patterns: the *one-from-many receive* and the *one-to-many send/receive*, as their representation most benefits from its introduction.

5.1 One-from-many receive

A party receives several related messages from autonomous events at different parties. Correlation of messages should occur within a time-frame. The number of messages may not be known at design or run-time [3].

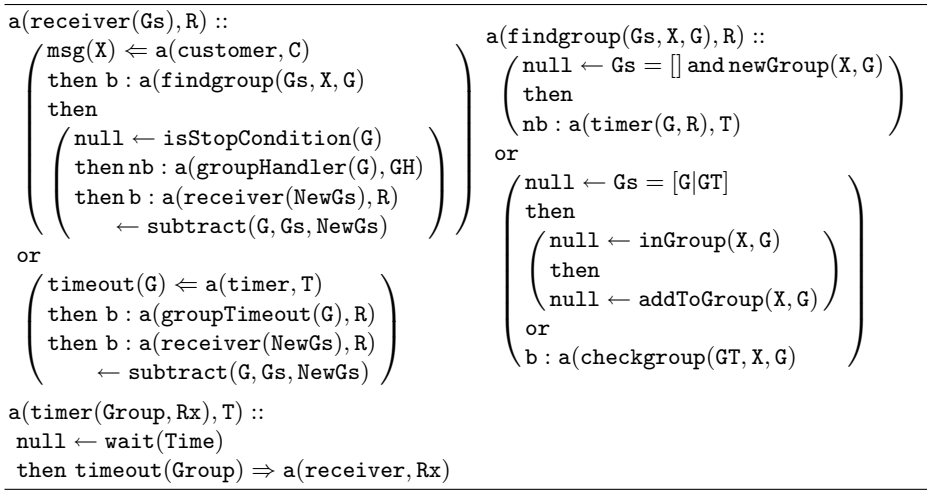

Figure 4. One-from-many receive implementation in LCC.

The solution is to create separate processes for each time-frame, for handling the proceeding after the stop condition, and using the central process for managing the reception and the dispatch of messages in groups. Figure 4 shows the implementation of the pattern using LCC. The `receiver` role is the recipient of the message about an event `X` from a `customer` or of a `timeout` message from a timer. When the message arrives, the peer in the `receiver` role first finds the group corresponding to the event `X` using the auxiliary role `findgroup`. This role recurses through the list of groups: at each recursion the membership of `X` to the current group is checked. If the corresponding group is found, it is returned in the output parameter `G`. If no group is found, a new group is created and a new parallel role `timer` is spawned. Back to the main role receiver, if the group is in `stopCondition` (sufficient messages have been received), a new role is spawned for handling the group. This role (not listed here) could contain the reference to a new scene that all the senders should perform. If a timeout message is received, the receiver takes the `groupTimeout` role. The role is not listed, but we assume a message is sent to every sender in the group to inform them of the timeout.

5.2 One to many send/receive

A party sends a request to several parties. Responses are expected within a time-frame and some parties may not respond. The number of parties may not be known at design time and the responses need to be correlated to their request [3].

Sending the requests is handled easily using the original LCC and a pattern similar to the one used in the auction protocol by the `waiter` role. The second part is difficult and clumsy to represent using the original LCC syntax. Using the non-blocking call we can write compact code, as it is shown in Figure 5.

```
a(caller(Rs), D) ::
 null ← Rs = []
 or
 ( (nb : a(invoker(R), C) ← Rs = [R|RT])
 then b : a(caller(RT), D) )
```

```
a(invoker(R), C) ::
 msg(X) ⇒ a(remote, R)
 then nb : a(timer(T, C), Z)
 then
 ( reply(Y) ⇐ a(remote, R)
 or timeout() ⇐ a(timer, Z) )
```

Figure 5. One to many send/receive implementation in LCC.

The peer in the `caller` role recurses over a list of recipients and spawns a new role `invoker` for each message to be sent. The `invoker` role, initialised with the identifier of the remote peer to contact, sends the message to the peer, spawn a `timer` role for the timeout and then waits for the reception of either the reply from the remote peer or a timeout from the timer process.

6 Related Work

There are relatively few languages targeted specifically at service choreography, for a survey refer to [2][1]. The most widely known are:

• WS-CDL [9] or the Web Services Choreography Description Language is the proposed W3C standard for service choreography. WS-CDL has native constructs supporting the functionalities provided by the extensions described in this paper, that is a mechanism for dealing with complex interactions and support for parallel operations. A package in WS-CDL can contain more than a single choreography. The `perform` activity can be used to launch other choreographies sharing variables between the caller and the called choreography. It also has a specific `<parallel>` construct.

• Let's Dance [10] is a language that supports service interaction modelling both from a global and local viewpoint. In a global (or choreography) model, interactions are described from the viewpoint of an ideal observer who oversees all interactions between a set of services. Local models, on the other hand focus on the perspective of a particular service, capturing only those interactions that directly involve it. Let's Dance supports parallel operations with a specific operator for repeated interactions; it does not provide a mechanism for abstraction.

• BPEL4Chor [4] is a proposal for adding an additional layer to BPEL to shift its emphasis from an orchestration language to a complete choreography language. BPEL4Chor is a collection of three artifact types: participant behaviour descriptions, participant topology and participant groundings. In BPEL4Chor it is possible, using an attribute, to define repeated interactions as parallel. It does not provide mechanisms for abstraction.

7 Conclusions

OpenKnowledge and LCC have been deployed in various scenarios, such as bioinformatics, emergency response simulation and health informatics. The extensive use (more than 200 different interaction models) has highlighted some of the limitations of the language and of the framework. In this paper we have analysed two limitations encountered in designing interactions: the lack of an abstraction mechanism to separate different levels of detail and the lack of a parallel operator.

In order to address these limitations, this paper proposed two extensions to LCC. The first is the introduction of the concept of scenes: a scene is an abstraction of an interaction model. The second is the introduction of a non-blocking role invocation, that allows the creation at run-time of new processes performing roles. The evaluation has shown that the new extensions allow a cleaner representation of service interaction patterns.

References

1. A. Barker, P. Besana, D. Robertson, and J. B. Weissman. The Benefits of Service Choreography for Data-Intensive Computing. In *Proceedings of CLADE '09*, pages 1–10. ACM, 2009.
2. A. Barker and J. van Hemert. Scientific Workflow: A Survey and Research Directions. In R. Wyrzykowski and et al., editors, *Seventh International Conference on Parallel Processing and Applied Mathematics, Revised Selected Papers*, volume 4967 of *LNCS*, pages 746–753. Springer, 2008.
3. A. Barros, D. M, and A. ter Hofstede. Service Interaction Patterns. In *BPM 2005*, pages 302–318. Springer, 2005.
4. G. Decker, O. Kopp, F. Leymann, and M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *Proceedings of IEEE ICWS 2007*, pages 296–303. IEEE Computer Society, July 2007.
5. D. Robertson, C. Walton, A. Barker, and P. Besana et al. Models of Interaction as a Grounding for Peer to Peer Knowledge Sharing. *Advances in Web Semantics I: Ontologies, Web Services and Applied Semantic Web*, 4891/2009:81–129, 2009.
6. N. Russell, A. ter Hofstede, W. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical Report BPM-06-22, BPM Center, 2006.
7. R. Siebes, D. Dupplaw, S. Kotoulas, A. P. de Pinninck, F. van Harmelen, and D. Robertson. The OpenKnowledge System: An Interaction-Centered Approach to Knowledge Sharing. In *Proceedings of CoopIS*, 2007.
8. G. Trecarichi, V. Rizzi, L. Vaccari, M. Marchese, and P. Besana. OpenKnowledge at Work: Exploring Centralized and Decentralized Information Gathering in Emergency Contexts. In *ISCRAM 2009*, 2009.
9. W3C. Web Services Choreography Description Language Version 1.0. http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/, November 2005.
10. J. M. Zaha, A. Barros, M. Dumas, and A. ter Hofstede. Let's Dance: A Language for Service Behaviour Modeling. In *OTM 2006*, volume 4274 of *LNCS*, pages 145–162. Springer, 2006.